Data Management for SSDs for Large-Scale Interactive Graphics Applications

Behzad Sajadi *
Shan Jiang †
M. Gopi‡
Department of Computer Science
University of California, Irvine

Jae-Pil Heo §
Sung-Eui Yoon ¶
Korea Advanced Institute of Science and Technology

Figure 1: *A few snapshots of our interactive out-of-core walkthrough scene editing on The City Model (consisting of 110M triangles). The first image from left shows part of the original scene; the second one shows the scene after selecting an object (the house shown in green in the first image is chosen and shown in blue in the second image); the third image shows the scene after duplication of the same object and the last one after removing the original copy of the object.*

Abstract

Solid state drives (SSDs) are emerging as an alternative storage medium to HDDs. SSDs have performance characteristics (e.g., fast random reads) that are very different from those of HDDs. Because of the high performance of SSDs, there are increasingly more research efforts to redesign the established techniques that are optimized for HDDs, to work well with SSDs. In this paper we focus on computing cache-coherent layouts of large-scale models for SSDs. It has been demonstrated that cache-oblivious layouts perform well for various applications running on HDDs. However, computing cache-oblivious layouts for large-models is known to be very expensive. Also these layouts cannot be maintained efficiently for dynamically changing models. Utilizing the properties of SSDs we propose an efficient layout computation method that produces a page-based cache-aware layout for SSDs. We show that the performance of our layout can be maintained under dynamic changes on the model and is similar to the cache-oblivious layout optimized for static models. We demonstrate the benefits of our method for large-scale walkthrough scene editing and rendering, and collision detection.

CR Categories: I.3.2 [Computer Graphics]: Graphics Systems—; I.3.8 [Computer Graphics]: Applications—;

Keywords: dynamic data layouts, cache-aware, cache-oblivious, cache-coherent layouts, out-of-core applications, walkthrough scene editing, walkthrough rendering, collision detection, solid state devices, solid state drives, flash drives, hard-disk drives.

*e-mail: bsajadi@uci.edu
†e-mail: sjiang4@ics.uci.edu
‡e-mail: gopi@ics.uci.edu
§e-mail: jaepil@jupiter.kaist.ac.kr
¶e-mail: sungeui@gmail.com

1 Introduction

Hard disk drives (HDDs) have been a very successful storage medium since the magnetic tape era. Thanks to their success, HDDs have inspired numerous research studies on various essential components of hardware and software including external data access models for out-of-core algorithms [Vitter 2001] and virtual memory schemes [Levy and Lipman 1982] for operating systems.

In addition to the general data management community, computer graphics and visualization researchers have also contributed tremendously to the management of graphics and visualization specific data using novel data structures, cache management techniques, and out-of-core algorithms for interactive rendering and many other graphics and visualization applications [Silva et al. 2002].

Recently, solid state drives (SSDs) (i.e. flash memory) are emerging as an alternative storage medium to HDDs. SSDs are widely adopted in mobile devices, laptops, and even desktop PCs, as the secondary storage. This phenomenon is mainly caused by their cheaper cost than DRAMs, and much faster access performance than HDDs [Agrawal et al. 2008].

SSDs have performance characteristics that are very different from those of HDDs. The random read performance of SSDs is much faster compared to HDDs. Moreover, the random read performance of SSDs is similar to their sequential read performance. Also, the sequential write performance of SSDs is similar to or even faster than their sequential and random read performance [Agrawal et al.

I3D 2011, San Francisco, CA, February 18 – 20, 2011.
© 2011 ACM 978-1-4503-0565-5/11/0002 $10.00

2008]. On the other hand, SSDs have performance that are relatively similar to DRAMs, except that SSDs are three times slower on average than DRAMs [1]. However, stored data in SSDs are non-volatile in contrast to DRAMs.

With the advent of SSDs, there are increasingly more research efforts to redesign the established techniques, that are optimized for HDDs, to work well with SSDs. Some examples include virtual memory hierarchy [Saxena and Swift 2009] and swapping system [Ko et al. 2008]. However, to the best of our knowledge, there has been no attempt to redesign various out-of-core graphics algorithms and external data management techniques to best leverage the characteristics of SSDs.

Main contributions: In this paper, we propose changes to the data management techniques from the interactive rendering standpoint to take into account the latest trends in the secondary storage. Especially, we reconsider the data layout problem that has been actively studied in the recent years. In summary this paper has the following contributions:

1. For geometric data, we observe that in case of SSDs, a simple cache-aware layout, that is easier and faster to compute than a cache-oblivious layout, provides the same performance as the cache-oblivious layout (Section 3).

2. Based on this observation, we propose a new dynamic cache-aware layout algorithm to maintain the layout while changing the data on the external drive. We adopt a simple dynamic kD-tree partitioning to merge, divide and update the partitions that preserve the required properties of the cache-aware layouts on SSDs (Section 4).

3. To provide a simple interface for applications to use this layout, we present an application-independent middleware that utilizes our dynamic cache-aware layout to perform the changes committed by the applications (Section 5).

In order to demonstrate the usefulness of our approach in practice, we present two applications that use our middleware as an abstract interface to the external drive (Section 6). The first one is an out-of-core walkthrough scene editing and rendering system that performs both read and write operations on the external drive through our middleware. The other one is out-of-core collision detection that is a read-only application with different access patterns, but uses the same data layout through our middleware. We demonstrate that the performance of these two applications on our dynamic cache-aware data layout is comparable to the performance of static offline cache-oblivious layouts on SSDs.

2 Related Work

In this section, we explain previous methods for computing cache-coherent layouts that can reduce the number of cache misses for various applications. However, most of these prior techniques have not been designed for SSDs, which have different characteristics compared to HDDs.

2.1 Data Layout Optimization

The order in which data elements are stored can have a major impact on the runtime performance. Therefore, there has been considerable effort on computing cache-coherent layouts of the data to match its anticipated access pattern. In computer graphics, rendering and processing sequences, and cache-aware and cache-oblivious layout techniques have been proposed. However, computing these layouts takes a lot of time and has not been widely used for dynamically changing meshes.

Rendering sequences: In the context of rendering, Deering [1995] pioneered the computation and use of layouts of triangles and vertices, called *rendering sequences or triangle strips*. Hoppe [Hoppe 1999] cast the computation of rendering sequences as a discrete optimization with a cost function. In this work the cost function for a specific vertex buffer size used in GPUs is calculated and used to compute layout. Since the layout is optimized for a particular cache parameter, it can be categorized as a cache-aware layout. More recently, Diaz-Gutierrez et al. [2005; 2006] presented a graph based algorithm for generalized cache-oblivious layouts of triangles for rendering and geometry processing.

Processing sequences: Isenburg et al. [2003] proposed processing sequences as a generalization of the rendering sequences for various kinds of large-data processing. This processing sequence can be stored as an indexed mesh, called *streaming mesh*. Streaming meshes are represented as interleaved triangles and vertices that can be streamed through a small buffer [Isenburg and Lindstrom 2005]. These representations are particularly useful for offline applications, e.g., simplification and compression, that can adapt their runtime computations to a fixed ordering. We present a similar indexed mesh storage format in which vertices may be duplicated across different pages, while a triangle is represented only once.

Cache-aware data layouts: There is a significant performance difference when data is accessed from register, L1 cache, main memory, or disk. Often, data access time from disk to main memory or main memory to cache is the major performance bottleneck of various applications. Therefore, it is natural to consider cache information when designing specific algorithms. Cache-aware layouts are constructed by directly using the knowledge of the cache parameters, such as cache block size, in computation of the layouts. Yoon et al. [2006] proposed a cache-aware layout algorithm by partitioning the data elements into different clusters which correspond to cache blocks in a particular cache.

Cache-oblivious data layouts: Many algorithms use space-filling curves [Sagan 1994] to compute cache-friendly layouts of grids. These layouts have been widely used to improve the performance of image processing [Velho and de Miranda Gomes 1991] and terrain visualization [Lindstrom and Pascucci 2001; Pascucci and Frank 2001]. However, space-filling curves are mainly used for grids, images, and volumes that have uniform structures. Yoon et al. [2005] proposed a generalized cache-oblivious layout of a mesh or a graph for efficient rendering and processing of massive models. Recently, Sajadi et al. [2009] proposed a graph-based cache-oblivious data layout scheme called the 2-factor layout and showed its benefits in a rendering method.

2.2 SSD-specific Data Handling

There have been many approaches for handling data for efficient use of the properties of the HDDs. In order to utilize the properties of the SSDs, some of the well-established components (e.g. virtual memory hierarchy [Saxena and Swift 2009]) that are designed for HDDs are reworked for better performance with SSDs. Since SSDs have a slow performance for random writes, write buffering techniques [Kim and Ahn 2008] employed in SSDs have been proposed. Also, because of the low write endurance in SSDs, various wear-leveling algorithms have been proposed [Agrawal et al. 2008]. Zhou and Meng [2009] proposed a conversion method that transforms random writes to a series of sequential writes, which are much faster than random writes.

[1] http://www.bitmicro.com/press_resources_flash_ssd.php

However, there have not been many techniques that leverage the properties of the SSDs to enable new computer graphics algorithms. In this paper we use various properties of the SSDs to enable out-of-core applications to use dynamically changing data (e.g. large scale editing of walkthrough scenes).

3 Data Layouts – Properties and Implications

It is well known that the seek time in HDDs during random reads/writes is substantial. Given an access pattern, linearly ordering the entire data in order to minimize the number of seeks is done by Cache-Oblivious Layouts (COLs). In contrast, having coherent data within individual small blocks of memory (usually the size of a cache page or disk page) reduces the number of pages that need to be read. Computing such small data coherent blocks is done using Cache-Aware Layouts (CALs). Since a layout that is coherent over the entire data (COL) is also coherent within small blocks, a COL can be used in place of a CAL, while the converse is not true.

Since CALs are only locally optimized, they can be computed in an efficient manner compared to COLs. Typically, COLs are computed based on the global optimization of the linear ordering of the entire data using the access pattern of the entire data. Because of the global optimality required by COLs, these layouts are usually computed using expensive graph algorithms, while CALs can be computed by faster clustering methods. In practice, computing COLs for polygonal models consisting of a few hundreds of millions of triangles can take a few hours. Moreover, an incremental layout computation for COLs is not known and thus COLs cannot be maintained efficiently for dynamic models.

Now let us consider the implications of using COL and CAL on specific external drive technologies when high dimensional spatial datasets are used, as is common in computer graphics and visualization applications. For such datasets, we prove in Section 3.1 that no disk layout can guarantee the coherency of parts of the data that are highly likely to be accessed together, when the data is stored in a linear layout on the disk. Therefore we cannot realize efficient access to data stored on HDDs.

On the other hand, since SSDs have no seek time constraints, the data only needs to be coherent inside each disk page and therefore a CAL without any global ordering performs similar to a COL. We validate this observation in Section 3.2.

3.1 Inefficiency of HDDs for High Dimensional Spatial Data Sets

In this section we show that any storage medium with slow random access (e.g., HDDs) cannot guarantee efficient data access for high dimensional spatial datasets. In order to prove this let us assume an N-dimensional spatial regular grid with K blocks along each dimension, totalling K^N blocks. Two blocks are considered adjacent if they share a face. We assume that adjacent blocks are more likely to be accessed together compared to nonadjacent blocks.

First we show that embedding a high dimensional dataset in a lower dimensional space (e.g. linear ordering) results in several gaps between the parts of data that are adjacent in the high dimensional space.

Lemma 1: For any N_1-dimensional embedding of the blocks where $N_1 < N$, the ratio of the adjacent pairs of blocks in the N-dimensional space which are also adjacent in the N_1-dimensional space to the total number of adjacent pairs in the N-dimensional space is at most

$$\alpha = (N_1 \times (K^N - K^{N-N/N_1}))/(N \times (K^N - K^{N-1})).$$

Proof. Using induction we prove that the number of adjacent pairs of blocks in any N-dimensional grid is

$$F(N,K) = N \times K^N - N \times K^{N-1}.$$

We use induction on N. The induction base is $N = 1$ where the number of adjacent pairs is clearly $K-1$. In the inductive step we assume that $F(N,K) = N \times (K^N - K^{N-1})$. Now let us consider an $(N+1)$-dimensional grid; it includes K N-dimensional grids. In addition to the adjacent pairs inside these grids, there are $(K-1)$ pairs of adjacent $(N-1)$-dimensional grids which include $(K-1) \times K^N$ pairs of adjacent blocks. Therefore

$$\begin{aligned} F(N+1,K) &= K \times F(N,K) + (K-1) \times K^N \\ &= (N+1) \times (K^{N+1} - K^N). \end{aligned}$$

We use the function $F(N,K)$ to compute α. The denominator of α is $F(N,K)$ and the numerator of α is $F(N_1, K^{N/N_1})$ since the number of blocks is still K^N but the grid in which the blocks are arranged is N_1-dimensional. Therefore

$$\begin{aligned} \alpha &= (N_1 \times (K^N - K^{(N/N_1)*(N_1-1)}))/(N \times (K^N - K^{N-1})) \\ &= (N_1 \times (K^N - K^{N-N/N_1}))/(N \times (K^N - K^{N-1})). \end{aligned}$$

□

When using disk layouts N_1 is always equal to 1. Therefore

$$\alpha = (K^N - 1)/(N \times (K^N - K^{N-1}))$$

, which goes towards $1/N$ for large values of K. Note that even for $N = 3$, α goes toward $1/3$ which means most of the adjacent blocks in the 3-dimensional space are not adjacent on the disk.

As an alternative to disk seeking from the current block to the next requested block, one can read all the blocks between the current block and the next requested one if their distance on the disk is small. The following lemma proves that such an approach that avoids the expensive disk seeking is also inefficient for high dimensional datasets.

Lemma 2: For any linear ordering of an N-dimensional grid of blocks there exist a pair of blocks that are adjacent in the N-dimensional grid but their distance in the linear ordering is greater than $(K^N - 1)/(N \times (K-1))$.

Proof. Let us find the maximum distance of two adjacent blocks in the linear ordering (β). We know that in the linear ordering there is a pair of blocks (the first and the last ones) that are $K^N - 1$ away from each other (as there are a total of K^N blocks). If these two blocks, the first and the last one, are adjacent to each other in the N-dimensional grid, clearly $\beta = K^N - 1$. But in the N-dimensional grid, if we have to go through a sequence of L (face-adjacent) blocks, and these blocks are distributed between the first and the last blocks in the linear ordering, the lower bound on the maximum distance (β) between any two of these L adjacent faces will be $(K^N - 1)/L$. In order to find a lower bound on β, we compute the maximum shortest distance between any two blocks in the N-dimensional grid (using the Manhatten distance metric). The two farthest blocks in any N-dimensional grid reside in the diagonally opposite corners of the grid, and there are $N \times (K-1)$ face adjacent blocks in the shortest path between them. Hence there exist a path between every two blocks where

$$L \leq N \times (K-1).$$

Therefore

$$\beta \geq (K^N - 1)/(N \times (K - 1)).$$

□

Note that since an N-dimensional grid includes K $(N-1)$-dimensional grids, we can also conclude that there are at least K pairs with distance greater than or equal to

$$(K^{N-1} - 1)/((N-1) \times (K-1)).$$

This proves that frequent seeks on the disk is unavoidable when dealing with high dimensional datasets. Therefore, HDDs, which provide slow random access, are inefficient for such datasets even though they can provide fast sequential access.

3.2 Experimental Results on Layouts

We proved that adjacency of the coherent blocks in a high dimensional space can not be preserved in a linear ordering. However, one of the main benefits of SSDs over HDDs is that the seek time is small and therefore the performance of random data access in SSDs is similar to that of their sequential data access, while random access in HDDs is much slower than the sequential access. To quantitatively verify this in a graphics application, we implement a walkthrough navigation system and test it with a large-scale city model (Figure 1) that consists of 110 M triangles, and occupies 3.7 GB of memory space that does not fit in the 1 GB of available main memory of the machine used for the experiments. Therefore, the walkthrough system runs in an out-of-core mode with the data stored in an external drive like a HDD or a SSD.

Page-based data structure: We use a self-contained page data structure as an atomic data access unit stored in the external drive. Each page has a fixed size (e.g., typical 4 KB pages [Chen et al. 2002], [Graefe and Larson 2001]) and contains a small but spatially coherent portion of the data. Each page consists of a set of triangles and only the vertices (and attributes) that are referred by those triangles. Since a vertex may be shared by multiple triangles in different pages, the shared vertex and its attributes may be repeated over different pages. However, there is no duplicate triangle in this page-based representation. This simple page-based data structure has been shown to work well for large-scale rendering [Sajadi et al. 2009].

Cache-oblivious and cache-aware data layouts: We first construct a COL of the model [Yoon and Lindstrom 2006]. From this COL, we construct the page-based COL, which is just the given COL organized in the page-based data structure without changing the sequence of the primitives, i.e. triangles. We process the triangles in the order they appear in the COL and group as many of them as possible, together with their vertices, in the self-contained fixed-size disk pages. The ordering of the triangles in the given COL implicitly determines the ordering of the pages. As a next step, from this page-based COL, we compute a page-based CAL by randomly shuffling the pages of the page-based COL to destroy the ordering of the pages. However, the primitives within each page are ordered in a cache-coherent manner for both layouts and the layout is considered as optimized for a particular disk page (or block) size.

Experiments on SSDs and HDDs: We measure the running time of our walkthrough scene rendering system using the two aforementioned disk layouts with a SSD and a HDD, under a fixed-path navigation of the scene. We consider for scenarios, the COL on the SSD, the COL on the HDD, the CAL on the SSD, and the CAL on the HDD. In particular, we measure the maximum delay between every two consecutive frames. The statistics are gathered in 100 frames intervals. Figure 2 shows the graphs of these tests.

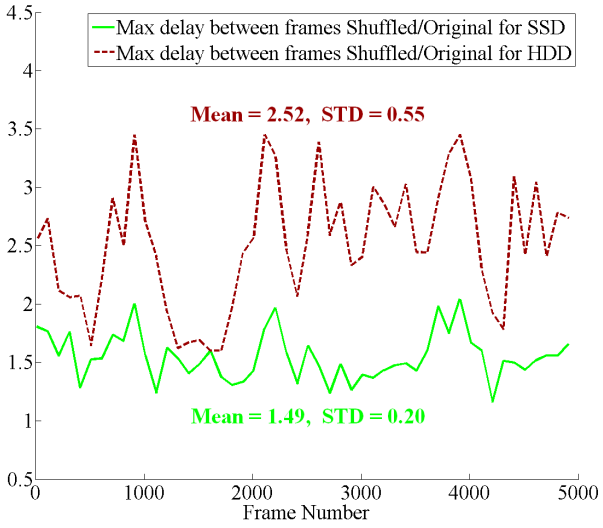

Figure 2: *The graphs show the ratio of the maximum delays between two consecutive frames after and before shuffling of the disk pages. The statistics are gathered in intervals of 100 frames during a walkthrough navigation of the City model. The high values in the graph for the HDD show that its performance degrades considerably after shuffling of the disk pages while the values for the graph for the SSD are small which show the SSD performance only slightly degrades after shuffling of the disk pages.*

Based on these graphs, we observe the following results:

1. With the HDD, the maximum delay values between the consecutive frames with the CAL are considerably higher compared to the COL layout. Many times, the maximum delay using the CAL is more than three times of the maximum delay using the COL, and more importantly it is never below the maximum delay for the COL. The maximum delays values are of high importance since long delays do not allow a smooth navigation with a uniform speed through the scene.

2. With the SSD, the maximum delay values are almost 1.5 times larger for the CAL compared to the COL. Further, the maximum delays values for the CAL are at most twice larger than the ones for the COL. More importantly, these measures are clearly better than those for HDDs. The 1.5 factor is due to the small page size used in our experiments which is not optimized for the SSD used in the experiments.

Based on these simple tests, we can conclude that the inter-page order captured in the COL is not critical when dealing with SSDs and CALs perform almost as good as COLs. This is an important observation because compared to COLs, CALs are easy to compute and maintain when the scene geometry changes. Moreover, there is no known incremental layout algorithm that supports efficient dynamic modifications for COLs. As a result, COLs are nearly impossible to be used for applications that frequently and dynamically change the model and commit to the external drive.

Based on this observation, in this paper we propose a novel page-based CAL for SSDs with similar performance compared to the COL for both SSDs and HDDs. More importantly, since the pages in the CALs can be placed anywhere on SSDs, our page-based CAL can be easily and efficiently maintained when performing dynamic modifications in the scene data. To show the efficiency of our layout, we use it in an out-of-core interactive scene editing application which to the best of our knowledge is the first of its kind.

4 Layout Construction Method

In order to construct the layout, we first need to compute clusters of the spatial data such that each cluster is spatially coherent, with a small bounding box. The clusters should be almost equal-sized and each fit into one page of the external drive. Finally, our layout construction method should run fast enough to handle dynamic changes at runtime. Given these page-sized clusters, the final cache-aware page-based layout is computed by concatenating the pages of the clusters in an arbitrary order.

One can use general clustering methods like the Lloyd's algorithm or spatial clustering methods like octree-based methods. However, these clustering methods do not guarantee clusters with equal size in terms of the number of triangles. Therefore, some of the clusters will end up being too large or too small to fit into a fixed-size disk page. Further, Lloyd's method is an iterative approach and is too slow for our purpose of interactive scene editing. In the case of octree, the positions of the partitioning planes and the number of children are fixed. Therefore, the lower bound on the resulting cluster size at the leaf nodes cannot be controlled precisely, resulting in a lot of sparsely populated pages and hence wasting the disk space. This also increases the number of pages required to store the data, and hence the size of the associated meta-data such as the page IDs and bounding box information, etc. As a result, this can result in poor in-core memory management for the application that uses the layout.

Instead, we propose to use a kD-tree to partition the data into clusters with roughly equal number of triangles. We recursively divide the kD-tree nodes until all the triangles assigned to each node of the tree fit in a page. The divisions happen alternately in all the spatial dimensions and the frequency of the divisions along each plane is proportional to the extent of the bounding box of the whole model along that dimension. Further, in order to guarantee that the pages are at least half full, we always choose the dividing planes such that the number of triangles on both sides are almost equal. We will show in Section 4.1 that we can guarantee this property, even after performing dynamic changes to the dataset, resulting in fewer pages to store the same amount of data, smaller in-core data, and hence better performance of the applications. The same criteria can not be guaranteed with octree subdivision and the kD-tree subdivision is also almost as fast as octree subdivision. The downside of using kD-tree subdivision is that it may produce spatially wide clusters in one dimension. However, we found that kD-tree subdivision performs reasonably well on our large models.

Our construction method for cache-aware page-based layouts is much faster than construction methods for cache-oblivious layouts. For example, in our experiments, the computation time for the out-of-core cache-oblivious data layout proposed by Yoon et al. [Yoon and Lindstrom 2006] was about one and a half hours for the City model consisting of 110 M triangles (Fig. 1), while the computation time for our layout was only 10 minutes. More details of the model are provided in Section 6.1.

4.1 Handling Dynamic Changes

In order to handle dynamic changes in a scene, we perform local reclustering on the dataset. We consider two types of dynamic changes: deletion and addition. Other operations such as moving or updating parts of the scene can be considered as a sequence of deletion and addition operations. As we construct an initial layout for a mesh with kD-tree subdivision, we also use kD-tree subdivision to handle such dynamic changes as follows.

Without loss of generality we assume that each add or delete operation is spatially local or can be divided to several such operations.

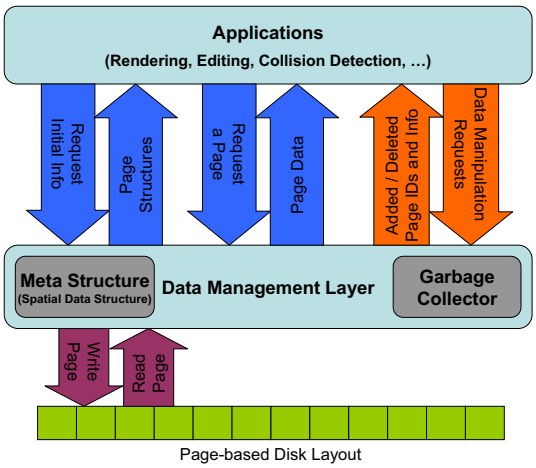

Figure 3: *Schematic view of our middleware, the layout of the underlying data, and applications*

In order to find the affected pages after any deletion or addition operation (i.e. pages that contain the deleted triangles or pages that are suitable to fit the new triangles) we keep an in-core kD-tree structure on the bounding boxes of the pages and check which leaf nodes of the kD-tree contain one or more of the newly added or deleted triangles. Each leaf node of the kD-tree contains pointers to a limited number of the pages (8 to 16 in our experiments) that their bounding boxes are inside, or overlap with, that node. We perform a local reclustering with the union of all the triangles in the pages referenced by these leaf nodes. A simple recursive kD-tree partitioning is performed *only* on these triangles. We partition the triangles until size of each partition is smaller than a disk page. Such a partitioning ensures that all the disk pages are at least half-full after local reclustering. The portioning planes are chosen with similar criteria described in Section 4 In Section 6, we demonstrate that this simple local reclustering works well even after several addition and deletion operations.

After local reclustering, the kD-tree structure on the bounding boxes of the pages will be updated by removing the old bounding boxes and adding the new ones. Also some of the sparse kD-tree nodes will be merged and the over-populated ones will be divided after every operation. In our experiments we chose to divide the nodes in the lowest level of the kD-tree when the number of the pages references by the node exceeds 16 and also we merge two sibling nodes when the total number of pages references by both the siblings is less than 16. Since the number of page bounding boxes is much smaller than the number of primitives, this kD-tree structure is small and can be stored and updated in the main memory efficiently.

5 Data Management Middleware for SSDs

In order to use the proposed disk layout for interactive graphics applications that generate interactive read and write requests on large out-of-core 3D geometric datasets, we propose a novel data management middleware specifically designed to be an interface between the external drives (preferably SSDs) and the graphics applications. The goal of this middleware is to maintain the performance of the graphics applications in spite of the dynamic changes in the dataset. Therefore, various applications achieve interactive performance without explicitly managing the underlying data. We achieve this goal by designing the middleware that uses and main-

tains our cache-aware layout of the underlying data on the external drive and provides an abstract interface for the applications to access and modify it. Our middleware also maintains a garbage collector in order to reuse the deleted data (i.e. pages) for later use. A schematic view of our middleware, data layout, and applications is shown in Figure 3.

We assume that the data is stored in the external drive in our self-contained page format as explained in the previous section. The middleware assigns a logical ID to each disk page and maintains a table to map these IDs to the physical location of the pages on the external drive. Each page also has an associated bounding box determined by the geometric primitives contained in that page. The bounding box information of all the pages are stored separately in the main memory and used along with the logical page IDs to provide an abstract interface between the middleware and the applications. All queries from the application to the middleware is processed through the identification of the logical page ID and/or the bounding box information.

The abstract interface includes the following main functionalities:

1. GETINITIALINFO: This initialization query to the middleware returns the list of logical IDs of all the pages together with their bounding box information.

2. FETCHPAGE: This query to the middleware, with the pageID as the input, finds the physical location of the page referenced by the pageID from the mapping table, reads the content of the page from the external drive and returns the content to the application.

3. DELETETRIANGLES, ADDTRIANGLES: These requests to the middleware have the input argument of the list of triangle IDs (along with their page IDs) that have to be removed or added. The middleware performs the edit operations on the given triangles, reclusters the local geometry information to maintain the layout, updates the kD-tree and the mapping table, and returns to the application a list of invalidated page IDs (of the deleted pages) and a list of new page IDs along with their bounding box information. The middleware also performs the garbage collection of all the deleted pages for future use in order to reserve coherent pages in the physical drive for the data from the same application.

The abstract page-based interface helps the applications to access the data efficiently without dealing with the arrangement of the data on the disk. Further, use of the bounding boxes of the pages instead of the triangles as the basic geometric elements keeps the number of elements small making it possible for the applications to manage the data with in-core data structures. Applications can use acceleration hierarchies such as octrees or kD-trees on the bounding boxes of the pages. Since each page has triangles from a spatially coherent part of the mesh, the bounding box of the triangles in a page is expected to be tight.

6 Applications and Results

We presented a middleware that provides an abstract interface for computer graphics and visualization applications and enables more applications to run interactively with dynamic large-scale models on SSDs using our novel CAL. To show the benefits of our method, we implemented two different applications that use the abstract interface of the middleware to interact with the model: out-of-core interactive walkthrough scene editing system and out-of-core collision detection system.

For all the experiments we used a Western digital HDD, model number WD1600HLFS, with 160 GB capacity, 16 MB cache space, 10000 RPM rotational speed, and SATA II 3.0 Gbps interface. We also used a Patriot SSD, model number PE64GS25SSD, with 64 GB capacity, 64 MB cache space, 210 MB/s read speed, 150 MB/s write speed, and SATA II 3.0 Gbps interface. In all the experiments we only measure the performance of different disk layouts on either the SSD or the HDD. The performance of the SSD is not directly evaluated against the HDD because of the differences between the speed and cache size of the drives.

6.1 Walkthrough Scene Rendering and Editing

We have developed an *out-of-core interactive walkthrough scene rendering and editing system* that allows the user to interactively delete, insert, copy, cut, and paste objects into and from the walkthrough scene. This was not possible earlier since editing a scene would destroy a good cache-oblivious layout that is required to achieve high performance with HDDs. Dynamically updating the layout while maintaining the cache-obliviousness is shown to be very difficult, if not impossible [Yoon and Lindstrom 2006]. Such a scene editing application is possible with SSDs since cache-aware layouts without any global ordering can be used efficiently with SSDs, and our novel layout computation method can update the layout at interactive rates while maintaining the cache-coherence. Hence to the best of our knowledge, this is the first interactive scene editing system for gigantic walkthrough models.

In addition to handling operations in the triangle level our scene editing application also provides the concept of objects in the scene, hence users can interact with such objects (e.g., a building or a bike). Each object is defined as a connected set of triangles. The scene editing operations can be applied similarly on an object or a set of triangles. Figure 1 shows results of copy/paste, and delete operations with the city model that consists of over 110 million triangles and occupies over 3.5 GB of the secondary storage in an uncompressed binary format and 3.7 GB in the page-based format. The total number of vertices is over 117 million in the original format and over 124 million after conversion to the page-based format. More details about the model and the page format can be found in our previous work [Sajadi et al. 2009].

6.1.1 Interacting with the Middleware

We use the abstract interface of the middleware in order to read and write the information from and to the disk. Initially the application uses the GETINITIALINFO function to get the bounding box information and page IDs from the middleware. The application constructs an in-core spatial acceleration hierarchy (i.e. an octree) on the bounding boxes of the disk pages. This acceleration hierarchy is used in each frame, to find the IDs of the disk pages visible in the view-frustum. After finding the page IDs, the pages are requested from the middleware using the FETCHPAGE function. The primitives inside the pages, returned by the middleware, are rendered and used for scene editing.

When a scene editing operation is performed, the application computes the affected triangles and the IDs of the pages that contain those triangles. Then, the editing operations are executed using the DELETETRIANGLES function or the ADDTRIANGLES function. The middleware performs the modification request and updates the data on the disk and returns the IDs of the invalidated pages and also the IDs and bounding box information of the new pages. The application updates its acceleration hierarchy by removing the invalid IDs and bounding boxes and adding the new ones.

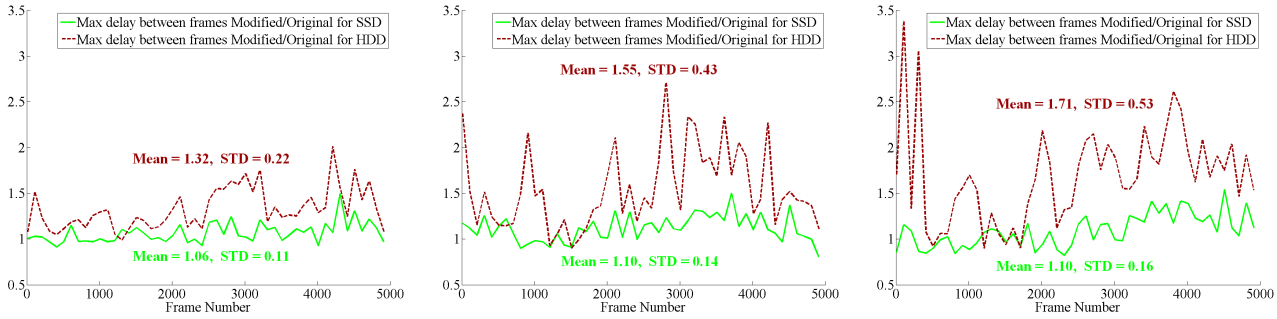

Figure 4: *The graphs show the ratio of the maximum delays between consecutive frames before and after interactive modification of 20%, 40%, and 60% of the scene from top to bottom. The statistics are gathered in time intervals of 100 frames during a walkthrough navigation on the City model. The dashed red graphs demonstrate that the performance of our dynamic CAL on the HDD degrades as more parts of the scene are modified; the maximum and variation of the values of the dashed red curves increase from the left graph for 20% modification to the right graph for 60% modification. However, the green curves demonstrate that on the SSD the performance changes very slightly and is comparable with the original COL, evident from the mean of the green curve which is only slightly above* 1 *and the maximum of it which is less than* 1.5*, even after modifying 60% of the scene.*

6.1.2 Performance Analysis

To demonstrate the efficiency of our CAL and middleware in handling dynamic changes for out-of-core scene rendering and editing, we started from a page-based COL of the city model and progressively modified 20%, 40% and 60% of the model using our scene editing tool to ensure that most of the data is ordered on the external drive using our dynamic CAL. We also made sure that the total number of triangles after the scene editing is similar to the original model. We gathered statistical information for the walkthrough rendering performance on the new scene while the data is stored on the SSD. We also copied the original and modified data to the HDD and gathered the same statistics. The results are shown in Figure 4. These experiments demonstrate that the performance of the application with our page-based CAL on the SSD does not degrade significantly even after changing 60% of the scene. However, with the HDD, the performance of the application progressively degrades with increasing changes from the original layout (please see the accompanying video). This confirms that our layout can handle dynamic changes in the data without significant degradation of the performance when used on SSDs but the same is not possible with HDDs. It is important to note that cache-oblivious layouts work well with both HDDs and SSDs, but are hard to maintain in an efficient manner during interactive scene editing. Therefore, using cache-oblivious layouts is not a practical solution for scene editing applications irrespective of whether it is for HDDs or SSDs.

6.2 Collision Detection

We also implemented a collision detection application to show the wide applicability of our method. We simulate the collision of several rocks falling on top of a castle and perform collision detection between each rock and the model (please see the accompanying video). The scene consists of over 30 million triangles and occupies more than 1.2 GB on the secondary storage (Figure 5).

In order to perform collision detection, we compute spatial hierarchical data structures (e.g., octree) for each model. We check whether there is an overlap between bounding boxes of two root nodes of two models. If there are overlaps, we traverse the hierarchy to localize the collisions. Once we reach the leaf nodes, we perform triangle-level collision detection [Ericson 2004]. Since contacts among objects occur in localized regions of those objects, only small portions of the hierarchy are accessed during the hierarchical traversal.

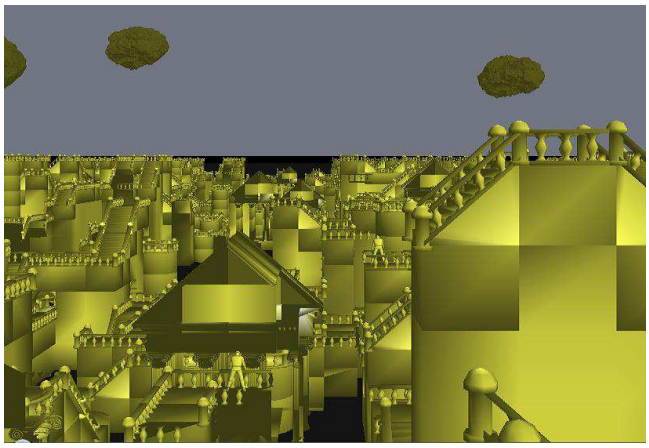

Figure 5: *A view of the castle model during collision simulation.*

The collision detection application uses the data management layer to read the pages from the SSD. We perform collision detection on our benchmark scene after performing several scene editing operations on the model using our out-of-core scene editing application described previously. The detailed performances of the HDD and the SSD before and after modification of the scene is demonstrated in Figure 6. In this chart, the average collision detection time for every 100 frames is used to gather the statistics.

This confirms that the performance of our dynamic layout does not degrade with several modification operations on the scene when used with SSDs but again performs poorly on HDDs. Therefore our layout can be used for other applications such as collision detection without any reordering of the data when using SSDs.

7 Conclusion

In conclusion, the impartial performance of SSDs with respect to cache-aware and cache-oblivious layouts has opened up new types of graphics applications which were inefficient or even impossible with HDDs. Specifically, to the best of our knowledge, we have demonstrated the first out-of-core interactive walkthrough scene editing tool which is not possible with HDDs since computing dynamic cache-oblivious layout of the data at interactive rates is difficult, if not impossible. In the process, the characteristics of SSDs

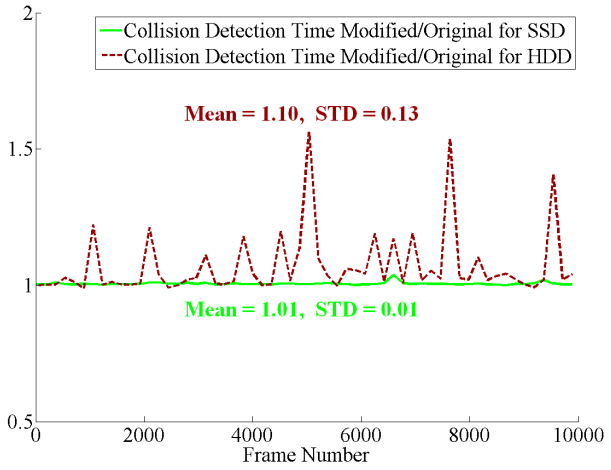

Figure 6: *The graphs show the ratio of the average collision detection times every 100 frames before and after interactive modification of 60% of the scene.*

have also lead us to the investigation of new techniques for dynamic clustering for cache-aware page-based layouts. We hope to see more novel applications and algorithms inspired by the characteristics of SSDs. In future we would like to try more applications on our data management middleware. We also want to try different clustering methods and more sophisticated techniques to achieve higher disk access performance on SSDs.

Acknowledgements

This research is partially funded by NSF grant CCF-0811809. Sung-eui Yoon was supported in part by MKE/MCST/IITA [2008-F-033-02], MKE digital mask control, KRF-2008-313-D00922, KMCC, MSRA, BK, DAPA/ADD (UD080042AD), MKE/KEIT [KI001810035261], and MEST/NRF/WCU (R31-2010-000-30007-0).

References

AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M., AND PANIGRAHY, R. 2008. Design tradeoffs for ssd performance. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, USENIX Association, Berkeley, CA, USA, 57–70.

CHEN, S., GIBBONST, P. B., MOWRY, T. C., AND VALENTIN, G. 2002. Fractal prefetching b+-trees: Optimizing both cache and disk performance. In *ACM SIGMOD International Conference on Management of Data (Madison, WI, 0306 June 2002*, ACM Press, 157–168.

DEERING, M. F. 1995. Geometry compression. In *ACM SIGGRAPH*, 13–20.

DIAZ-GUTIERREZ, P., BHUSHAN, A., GOPI, M., AND PAJAROLA, R. 2005. Constrained Strip Generation and Management for Efficient Interactive 3D Rendering. In *Proc. of Computer Graphics International Conference*, 115–121.

DIAZ-GUTIERREZ, P., BHUSHAN, A., GOPI, M., AND PAJAROLA, R. 2006. Single Strips for Fast Interactive Rendering. *The Visual Computer 22*, 6, 372– 386.

ERICSON, C. 2004. *Real-Time Collision Detection*. Morgan Kaufmann.

GRAEFE, G., AND LARSON, P. A. 2001. B-tree indexes and cpu caches. *Data Engineering, International Conference on 0*, 0349.

HOPPE, H. 1999. Optimization of mesh locality for transparent vertex caching. *ACM SIGGRAPH*, 269–276.

ISENBURG, M., AND LINDSTROM, P. 2005. Streaming meshes. *IEEE Visualization*, 231–238.

ISENBURG, M., LINDSTROM, P., GUMHOLD, S., AND SNOEYINK, J. 2003. Large mesh simplification using processing sequences. *IEEE Visualization*, 465–472.

KIM, H., AND AHN, S. 2008. Bplru: a buffer management scheme for improving random writes in flash storage. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, USENIX Association, Berkeley, CA, USA, 1–14.

KO, S., JUN, S., RYU, Y., KWON, O., AND KOH, K. 2008. A new linux swap system for flash memory storage devices. In *ICCSA '08: Proceedings of the 2008 International Conference on Computational Sciences and Its Applications*, IEEE Computer Society, Washington, DC, USA, 151–156.

LEVY, H. M., AND LIPMAN, P. H. 1982. Virtual memory management in the vax/vms operating system. *Computer 15*, 3, 35–41.

LINDSTROM, P., AND PASCUCCI, V. 2001. Visualization of large terrains made easy. *IEEE Visualization*, 363–370.

PASCUCCI, V., AND FRANK, R. J. 2001. Global static indexing for real-time exploration of very large regular grids. In *Supercomputing*.

SAGAN, H. 1994. *Space-Filling Curves*. Springer-Verlag.

SAJADI, B., HUANG, Y., DIAZ-GUTIERREZ, P., YOON, S.-E., AND GOPI, M. 2009. A novel page-based data structure for interactive walkthroughs. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, 23–29.

SAXENA, M., AND SWIFT, M. M. 2009. Flashvm: Revisiting the virtual memory hierarchy. In *Proc. of USENIX HotOS-XII*.

SILVA, C., CHIANG, Y.-J., CORREA, W., EL-SANA, J., AND LINDSTROM, P. 2002. Out-of-core algorithms for scientific visualization and computer graphics. In *IEEE Visualization Course Notes*.

VELHO, L., AND DE MIRANDA GOMES, J. 1991. Digital halftoning with space filling curves. In *ACM SIGGRAPH*, 81–90.

VITTER, J. S. 2001. External memory algorithms and data structures: dealing with massive data. *ACM Comput. Surv. 33*, 2, 209–271.

YOON, S.-E., AND LINDSTROM, P. 2006. Mesh layouts for block-based caches. *IEEE Trans. on Visualization and Computer Graphics (Proc. Visualization) 12*, 5, 1213–1220.

YOON, S.-E., LINDSTROM, P., PASCUCCI, V., AND MANOCHA, D. 2005. Cache-Oblivious Mesh Layouts. *ACM Transactions on Graphics (SIGGRAPH) 24*, 3, 886–893.

ZHOU, D., AND MENG, X. 2009. Rs-wrapper: random write optimization for solid state drive. In *CIKM '09: Proceeding of the 18th ACM conference on Information and knowledge management*, ACM, New York, NY, USA, 1457–1460.